Course-Grain Multithreading

Version 1.1

01/99

Order Number: 243636-002

Table of Contents

Revision History

Revision	Revision History	Date
1.1	FCS revision	01/99

References

The following documents are referenced in this application note, and provide background or supporting information for understanding the topics presented in this document. Please contact your Intel representative for information about the availability of particular application notes.

1 *Streaming SIMD Extensions -- Diffuse Directional Lighting*, Intel® Application Note AP-596, Order No: 243630.

2 *Streaming SIMD Extensions -- 3D Transformation*, Intel Application Note AP-597, Order No: 243631.

3 *Wiener Filtering Using Streaming SIMD Extensions*, Intel Application Note AP-807, Order No: 243641.

4 *Achieving High-Performance 3D Geometry on a Pentium® III Processor*, Intel Application Note AP-816, Order No: 243650.

1 Introduction

This application note presents one possible method for writing applications that take advantage of multiple processor systems, in combination with an operating system that supports multithreading. The method presented here is called "course-grain" multithreading.

Multithreading is viewed as either coarse-grain or fine-grain, depending on the level of abstraction from the processor. Coarse grain multithreading is handled by the OS and is usually separated at a function level (an example is a thread which prints a file). The OS continues operating on other tasks while the print job is in progress. Fine grain multithreading is handled by the processor and is usually separated at the instruction level. An example is switch-on-event multithreading where the processor executes instructions from other processes when an instruction from one process blocks, either for memory accesses or resource conflicts. This type of multithreading is part of the microprocessor architecture. For operability across multiple microprocessor architectures, this application note concentrates on architecture independent coarse-grain multithreading.

Many types of applications benefit from partitioning one or more parts of the workload among multiple processors. However, it is typical that the larger the data set to be processed, the more benefit can be gained. The example presented in this document is for 3D geometry processing, because the data sets can be quite large even for what might be considered a simple scene. The algorithm shows how to execute multiple copies of the geometry pipeline in parallel on separate processors.

Furthermore, 3D geometry processing can get additional benefit from using Streaming SIMD Extensions. Refer to the References section for a list of other relevant documentation.

In addition to providing a C++ code example that shows how to implement the course-grain multithreading model, this paper addresses the following issues/topics:

- How to determine whether your application could benefit from coarse-grain multithreading.
- Partitioning an application for coarse-grain multithreading.
- Data set selection.
- Performance considerations for various Operating Systems and Intel® microprocessors.

2 Coarse-grain Multithreading

Threads are independent paths of execution in a program with separate CPU and stack memory state; that is, they have their own stack and copy of the CPU registers (global variable data can be accessed by all threads in a program). Switching between threads incurs a task switching overhead which is dependent on the OS and CPU architecture. To minimize the effect of task switching, the user should process larger data blocks in each thread.

This application note characterizes multithreaded applications as having a common set of functions operating on independent data; there is no static data shared between threads. In addition, a two processor system is assumed in which the two copies of the common set of functions are running simultaneously and independently of each other.

One example of such an application is a 3D geometry pipeline for transforming, lighting, culling, and clipping 3D object data. A single main thread synchronizes the input data setup and execution sequence for each pipeline thread through the use of mutexes, or some other signaling mechanism such as semaphores.

A mutex, mutual exclusion, is an OS resource for communicating between asynchronously executing threads. Synchronization between threads is performed by requiring each thread to "own" the mutex resource prior to executing a critical section of code or operating on shared data. A thread gets ownership by waiting for and then locking the mutex resource. All other threads are blocked, waiting for the mutex, while this thread operates on the shared data. The thread releases ownership of the mutex after processing the shared data. The OS then gives ownership of the mutex to the next thread that is being blocked by the mutex. Multiple mutexes may be active simultaneously.

For more information on multithreading and the use of static data between threads, the reader should refer to the Microsoft Developer Network Library section on Win32 multithreading.

2.1 Applications for Coarse-grain Multithreading

Course-grain multithreading is applicable to algorithms that process large, independent data sets with a common set of functions. Some immediately obvious applications that benefit from multithreading are games and 3D content development. 3D scenes are comprised of multiple objects, each in their own local coordinate system, which are the output of a scene management layer and the input to a rasterization layer. A common geometry pipeline processes the data elements (vertices, normals, texture coordinates, etc.) in these individual objects. The data elements of one object are independent of those from another object. Inter-object dependencies, such as collision detection, are handled outside the geometry pipeline by the scene management layer.

2.2 Implementing Coarse-grain Multithreading

The multithreading framework described here is easily extended to other parallel algorithms in which a section of code is executed multiple times with different data for each pass. The OS thread overhead must be factored in when determining whether an algorithm or section of code is suitable to multithreading. The code must perform sufficient work to amortize the thread overhead to an acceptable level. For the 3D example discussed in this paper, objects with <200 vertices would see little benefit from multithreading the pipeline versus a single threaded version, because the overhead cost of setting up data structures and task switching dominates the processing time. Objects with >1000 vertices would see nearly a factor of two speed up (on a dual processor system) because the pipeline processing dominates the setup and task switching costs. Two approaches to minimizing thread synchronization overhead are:

1 Process only large single objects

2 Process groups of smaller objects stored in a queue structure, giving the appearance of a single large object.

2.2.1 Techniques

A 3D application illustrating a coarse-grain multithreaded geometry pipeline is shown in Figure 1. This application has an initialize phase and a looping phase. In the initialize phase, two threads are spawned; each executing a copy of the geometry pipeline.

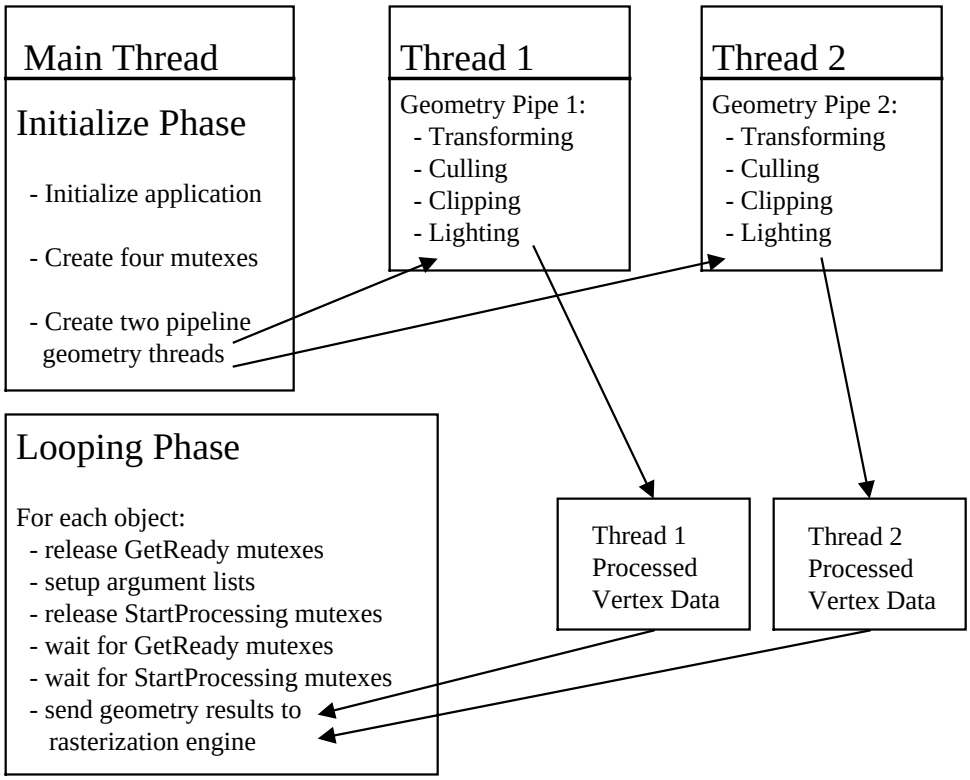

Figure 1: Geometry pipeline for course-grain multithreading

_beginthread(ProcessObject, 0, 0); // start copy 1 of geometry pipeline

_beginthread(ProcessObject, 0, 1); // start copy 2 of geometry pipeline

`ProcessObject` is the C function for a geometry pipeline. Four mutexes are initialized to handle handshaking between the looping process and the two geometry pipeline processes.

```
hGetReadyMutex[0] = CreateMutex( NULL, TRUE, NULL );
hGetReadyMutex[1] = CreateMutex( NULL, TRUE, NULL );
hStartProcessingMutex[0] = CreateMutex( NULL, TRUE, NULL );
hStartProcessingMutex[1] = CreateMutex( NULL, TRUE, NULL );
```

The four mutexes are initially owned by the main process; all synchronization starts in the looping phase of the main process.

Figure 2 shows a flow chart that illustrates the use of mutexes. In the looping phase, the main process releases a "Get Ready" mutex for each pipeline thread. After setting up data structures for each pipeline, the main process releases a "Start Processing" mutex for each pipeline, then waits for the "Get Ready" mutexes to be released by the two threads. This exchange of mutexes ensures that the main process and the two geometry pipeline threads are kept synchronized. While the pipeline threads are transforming their corresponding objects, the main process waits for the threads to release the "Start Processing" mutexes. After the pipeline threads release the “Start Processing” mutexes, the main process sets up the next object data structures, if a single object is processed by the thread, and these events repeat until all objects in the scene are processed. Another method is to group objects in queue structures prior to releasing the “Start Processing” mutexes. Each thread processes N/2 objects before returning control back to the main process. In this scenario, a single thread synchronization stage is sufficient to process

all objects. The main drawback of using the queue structure is that the main process must sort the objects and balance the workload presented to each thread. This incurs CPU overhead that is not in the single object per thread synchronization case. Complex objects with a large number of vertices and normals to process in the pipeline are ideal for this mechanism because the mutex and thread switching costs are amortized over a large number of processor cycles.

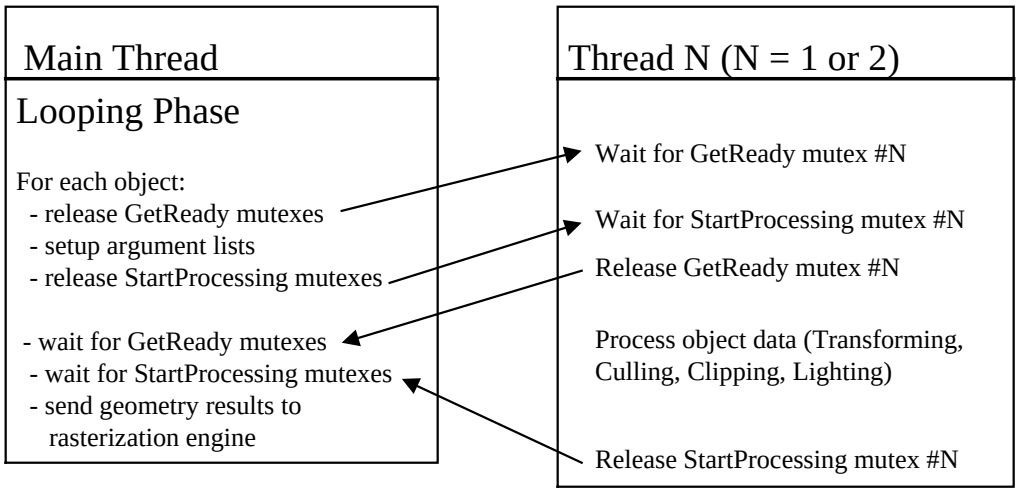

Figure 2: Flow chart illustrating the use of mutexes

3 Performance

The performance of a multithreaded application depends on factors related to the system configuration, CPU/cache speed and architecture, and the Operating System. An ideal multithreaded application balances the system memory bandwidth and the CPU processing performance. Memory bandwidth is the critical resource for the 3D object processing application used in this application note.

An attempt was made to minimize the amount of unused data per cache line by packing data structures such that contiguous bytes in each structure have temporal locality. Two 3D object processing threads were created and kept live throughout program execution. Mutexes were used to synchronize the starting and stopping of these threads from a main process thread. An alternate approach, which would have resulted in significant thread overhead, would have been to create and destroy new threads for each parallel 3D object processing task.

The data set size and L2/L1 cache sizes affect multithreaded and single-threaded performance differently, depending on the ratio of the data set size to the cache size. For data sets larger than the L2 and L1 cache, the caches are ineffective at exploiting data locality between 3D scenes for single-threaded and multithreaded applications. In this case, the performance depends only on the CPU to main memory bandwidth. For data sets smaller than the L2 and/or L1 cache, the caches may exploit temporal data locality for single-threaded applications (uni-processor configuration) and not for multithreaded applications (dual-processor configuration). In the dual processor case, data objects processed in one 3D scene may be processed on processor A, with data cached in processor A's caches, and then processed on processor B for the next 3D scene, thereby invalidating the data in processor A's caches.

3.2 Considerations

The multithreaded application outlined in this application note achieves the greatest performance benefit over a single threaded version for dual processor systems using a multithreaded OS (such as NT), large independent data sets, and application code that balances memory bandwidth with execution time. Clearly an OS such as Window 95, which doesn't support dual processors, will show degraded performance for these types of multithreaded applications because the application still has the thread overhead with mutexes without the benefit of concurrence. The data sets must be independent; otherwise, the two threads will compete for and cache the same resources, in effect decreasing the memory bandwidth. Memory bandwidth must be balanced with execution time in order to maximize the instructions per cycle for each thread. Concurrent operations are effective at increasing the overall application performance when actual work is being done. An imbalance between the memory and execution unit speeds results in no work being done while execution units wait for data from memory. In this case, prefetching data from memory can partially negate the disparity between the memory and execution unit speeds.

4 Conclusion

For algorithms with large, modular data sets, multithreading the section of code that processes these data modules significantly improves performance over single threaded models. In this case, a multithreaded geometry pipeline operating on multiple complex objects, with independent data per object, was described. Care must be taken to ensure synchronization between the main process and the parallel threads. Race conditions in which the main process does not wait for the threads to complete their tasks can occur without the proper use of mutexes. As the complexity of the objects (one metric is the number of vertices) decreases, task switching and data setup begin to dominate, negating the benefits of parallel threads.

5 C Coding Example

The code presented in this section is not a complete application that can be compiled and run on its own. This code is meant to show what is needed to implement multi-threading and should be incorporated into an application.

```
// A multithreaded geometry pipeline
// The scene being processed consists of multiple independent objects
// Interobject processing, eg. collision detection, is performed prior to
// the pipeline
// Two copies of the geometry pipeline are executing simultaneously
// Therefore, two sets of mutexes and argument lists are maintained for
// sychronization and communication between the main process and the two
// geometry threads.

// Define the object and argument passing structures
typedef struct Object_t {
  int           *pVertexIndex; // array of vertex indices describing
                               // either individual triangles or tri-strips
  Vertex        *pVertex;      // array of x,y,z,w components
  Color         *pVertexColor; // array of red, green, blue, alpha
```

```
  Normal        *pNormal;     // array of vertex normals in x,y,z
  TexCoord      *pTexCoord;   // array of texture coordinates in u,v
} Object;

typedef struct ArgList_t {
  Object        *pObject;     // pointer to an object or list of objects
  VertexInfo    *pOutBuffer;  // array of vertex, color, and texture data
                              // that is sent to rasterization engine
  int           ProcessDone;  // inter-thread flag to detect race
                              // condition between main thread and
                              // geometry thread
} ArgList;

// Global variables
HANDLE  hGetReadyMutex[2];        //
HANDLE  hStartProcessingMutex[2];
Arglist  Args[2];
Object   Object[N];               // N = total # of objects
Vertex   *OutputBuffer[N];        // One output buffer per object

// The following WinMain shows only the sections related to
// multithreading. Add your favorite Windows initialize code.

int WINAPI
WinMain (HINSTANCE hInst, HINSTANCE hPrevInst,
         LPSTR lpCmdLine, int nCmdShow)
{
MSG     msg;
HWND    hWnd;

hWnd = InitializeWindow (hInstance, nCmdShow);
Initalize_application_code( );

// Create the mutexes
// GetReady and StartProcessing mutexes are initially owned by main process
hGetReadyMutex[0] = CreateMutex( NULL, TRUE, NULL );
hGetReadyMutex[1] = CreateMutex( NULL, TRUE, NULL );
hStartProcessingMutex[0] = CreateMutex( NULL, TRUE, NULL );
hStartProcessingMutex[1] = CreateMutex( NULL, TRUE, NULL );

// Start two ProcessObject threads. The last argument in the _beginthread
// call specifies which copy of the geometry pipeline we're instantiating
_beginthread( ProcessObject,  0, 0); // start copy 1 of geometry pipeline
```

```
_beginthread( ProcessObject,  0, 1); // start copy 2 of geometry pipeline

// Main window message loop
while (GetMessage (&msg, NULL, 0, 0))
{
   TranslateMessage (&msg);
   // WM_PAINT messages are used here to activate the geometry pipeline
   // threads and recalculate the geometry for each object in the scene
   DispatchMessage (&msg);
}

// Clean up mutex handles
CloseHandle( hGetReadyMutex[0] );
CloseHandle( hGetReadyMutex[1] );
CloseHandle( hStartProcessingMutex[0] );
CloseHandle( hStartProcessingMutex[1] );

// Return success of application
return TRUE;
}

// The following MainWndProc shows only the sections related to
// scene redraw and geometry processing. Add your favorite message
// handling code.

// Window message handling procedure
LONG WINAPI
MainWndProc ( HWND hWnd, UINT uMsg,
              WPARAM wParam, LPARAM lParam)
{
   switch (uMsg)
   {
      case WM_PAINT:
      {
         DrawScene();
      }
      break;
   }
   return lRet;
}

// DrawScene sets up two argument structures with pointers to
// object data, and then triggers the processing of that data
```

```
// through the StartProcessing  mutex. The GetReady mutex ensures
// that DrawScene does not setup the next object until the current
// object has been fully processed by ProcessObject.

void DrawScene()
{
  ReleaseMutex(hGetReadyMutex[0]);
  ReleaseMutex(hGetReadyMutex[1]);

// Assumes an even number of objects
  for(obj=0; obj<NumberOfSceneObjects; obj+=2)
  {
    // Setup argument list first geometry pipeline thread
    Args[0].pObject =&Object[obj]; // next object in scene to process
    Args[0].pOutBuffer =&OutputBuffer[obj]; // results buffer
    Args[0].ProcessDone = 0;  // set flag to not done
    // Setup argument list second geometry pipeline thread
    Args[0].pObject =&Object[obj+1];  // next object in scene to process
    Args[0].pOutBuffer =&OutputBuffer[obj+1]; // results buffer
    Args[0].ProcessDone = 0;  // set flag to not done
    // release ownership of StartProcessing mutex – this will cause each
    // pipeline to start processing their respective object data
    ReleaseMutex(hStartProcessingMutex[0]);
    ReleaseMutex(hStartProcessingMutex[1]);
    // main process ensures that each pipeline thread gets execution time
    // by calling OS function WaitForSingleObject
    WaitForSingleObject(hGetReadyMutex[0], INFINITE );
    WaitForSingleObject(hGetReadyMutex[1], INFINITE );
    // main process waits for each pipeline thread to complete and release
    // ownership of their StartProcessing mutexes
    WaitForSingleObject(hStartProcessingMutex[0], INFINITE );
    WaitForSingleObject(hStartProcessingMutex[1], INFINITE );
    if(Args[0].ProcessDone != 1)
      return;                       // Race condition detected
    if(Args[1].ProcessDone != 1)
      return;                       // Race condition detected
  }
// All objects have been processed and the OutputBuffer arrays contain the
// processed results
// Send Vertex, Index, Color, Texture, etc. data stored in OutputBuffer
// arrays to rasterization engine
}
```

```
// ProcessObject receives a pointer to an Object structure through
// one of the Args structures. When the ProcessObject threads were
// started, a ThreadNum variable was passed in denoting which Args
// element to use. All pointers and data are passed through globally
// allocated structures.

void ProcessObject(void *ThreadNum)
{
// Need ThreadNum (0 or 1) to determine which global Args structure
// to use.

// Get pointer to object from global structure
pObject = Args[ThreadNum].pObject;
// Get pointer to output buffer
pOutBuffer = Args[ThreadNum].pOutBuffer;

// Perform geometry pipeline code here for transforming, lighting,
// clipping, culling, etc. the objects vertex and triangle data

Args[ThreadNum].ProcessDone = 1;  // set flag to done
// Release ownership of StartProcess mutex to signal main thread to continue
ReleaseMutex(hStartProcessingMutex[ThreadNum]);
}
```